[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 12: The Future of Data Systems

Posted by CHARLES *on* 2020-05-15

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

If a thing be ordained to another as to its end, its last end cannot consist in the preservation of its being. Hence a captain does not intend as a last end, the preservation of the ship entrusted to him, since a ship is ordained to something else as its end, viz. to navigation. (Often quoted as: **If the highest aim of a captain was the preserve his ship, he would keep it in port forever.**) — St. Thomas Aquinas, Summa Theologica (1265–1274)

- **Goal** for the book was: how to create applications and systems that are **reliable**, **scalable**, and **maintainable**.
- Our goal is to discover how to design applications that are better than the ones of today— **robust, correct, evolvable, and ultimately beneficial to humanity**.

Data Integration

- Common theme of the book: **for any given problem, there are several solutions**. (C: As it should be for anything in life)
- Challenges:
  - 1, mapping between tools and circumstances;
  - 2, in complex applications, data is often used in several different ways.
- **Combining Specialized Tools by Deriving Data**:
  - The need for data integration often only becomes apparent if you zoom out and consider the dataflows across an entire organization.
  - **Reasoning about dataflows**:
    - Need be very clear about the INs/OUTs of the data;
  - **Derived data vs. distributed transactions**:
    - At an abstract level, they achieve a similar goal by different means.
  - **The limits of total ordering**:

- **Batch and Stream Processing**:
  - The **goal of data integration** is to make sure that data ends up in the right form in all the right places.
  - There are also many detailed differences in the ways the processing engines are implemented, but these distinctions are beginning to blur.
  - **Maintaining derived state**:
    - Batch processing has a quite strong functional flavor.
    - Asynchrony is what makes systems based on event logs robust.
  - **Reprocessing data for application evolution**:
    - Reprocessing existing data provides a good mechanism for maintaining a system. (e.g. Schema Migrations on Railways, with three rails)
    - Derived **views** allow gradual evolution. (Because underlying data hasn't changed)
      - Every stage of the process is easily reversible if something goes wrong. (C: echo Jeff Bezos' type 1 or 2 decision)
  - **The lambda architecture**: combine stream and batch processing.
    - The lambda architecture proposes running two different systems in parallel:
      - a batch processing system such as Hadoop MapReduce,
      - a separate stream processing system such as Storm.
    - Potential problems:
      - Dual-maintenance;
      - Data need to be merged in order to produce results;
      - Reprocessing data is expensive on large datasets;
  - **Unifying batch and stream processing**:
    - allowing both batch computations (reprocessing historical data) and stream computations (processing events as they arrive) to be implemented in the same system.

Unbundling Databases

- At a most abstract level, **databases, Hadoop, and operating systems** all perform the same functions: they store some data, and they allow you to process and query that data.
  - at their core, both are "information management" systems
- Unix vs. Relationship Model
- **Composing Data Storage Technologies**:
  - Similar features that built for both "DBs" and "batch & stream processing";
  - **Creating an index**:
    - Whenever you run CREATE INDEX, the database essentially **reprocesses** the existing dataset and **derives** the index as a new view onto the existing data.
  - **The meta-database of everything**:

date; (regardless is batch, stream or ETL)
    - Two cohesive system in the future:
        - **Federated databases: unifying reads**; (e.g. PostgreSQL's foreign data wrapper)
            - Application still able to access low-level data engine;
            - User can grab combined data from the federated interface;
        - **Unbundled databases: unifying writes**;
- **Making unbundling work**:
    - **Federation** and **unbundling** are two sides of the same coin: composing a reliable, scalable, and maintainable system out of diverse components.
    - Synchronizing writes is definitely a harder problem to solve.
    - Asynchronous event log with idempotent write **over** distributed transactions.
    - log-based integration is loose coupling between the various components.
        - 1, at a system level, asynchronous event streams make the system more robust and high-fault tolerance;
        - 2, at a human level, unbundling data systems allows subsystems to be more elastic/maintainable.
- **Unbundled vs. integrated systems**:
    - Specialized query engines will continue to be important for particular workloads.
    - It's about breadth, not depth.
    - If there is one software/system that can satisfy all your needs, then use it. Unless there isn't one, then you should consider unbundling/composition.
- **What's missing**?
    - We don't yet have the unbundled-database equivalent of the Unix shell. (e.g. declare mysql | elasticsearch, by analogy to Unix pipes)
    - E.g. differential dataflow
- **Designing Applications Around Dataflow**:
    - "database inside-out" approach, more like a "design pattern" rather than a "new architecture"
    - **Goal**: when a record in a database changes, we want any index for that record to be automatically updated, and any cached views or aggregations that depend on the record to be automatically refreshed. (And you don't need worry about the technical details)
    - **Application code as a derivation function**:
        - When one dataset is derived from another, it goes through some kind of transformation function. (e.g. secondary index, full-text search, ML system, cache)
        - Custom code within the database is the part where most of them struggle.
    - **Separation of application code and state**:
        - some parts of a system that specialize in durable data storage, and other parts that specialize in running application code. (e.g. Web application

    - not putting application logic in the database and not putting persistent state in the application
    - DB acts as a kind of **mutable shared variable** that can be accessed synchronously over the network.
  - usually you can't subscribe to changes in a mutable variable, you can only read it periodically. (e.g. unlike Spreadsheet)
    - But you can implement your own notification —> aka. Observer pattern.
    - (C: this subscribe model also used by the Flutter state management package "Provider", it in a way of subscribe change and notify the UI to render)
    - Similar approach goes to DBs (subscribing to change only just beginning as a feature)
- **Dataflow: Interplay between state changes and application code**:
  - Application code responds to state changes in one place by triggering state changes in another place.
  - (C: This "unbundling" approach is very similar on the talk I watched that built systems around Apache Kafka)
  - Maintaining derived data is not the same as asynchronous job execution
    - 1, the order of state changes is often important.
    - 2, fault tolerance is the key
- **Stream processors and services**:
  - service-oriented architecture(SOA) over a single monolithic application is primarily organizational scalability through loose coupling.
  - Composing **stream operators** into dataflow systems has a lot of similar characteristics to the microservices approach.
    - But, it is one-directional, asynchronous message streams rather than synchronous request/response interactions.
  - "**The fastest and most reliable network request is no network request at all!**" (C: so make everything as local as possible)
- Subscribing to a stream of changes, rather than querying the current state when needed, brings us closer to a spreadsheet-like model of computation:
- **Observing Derived State**:

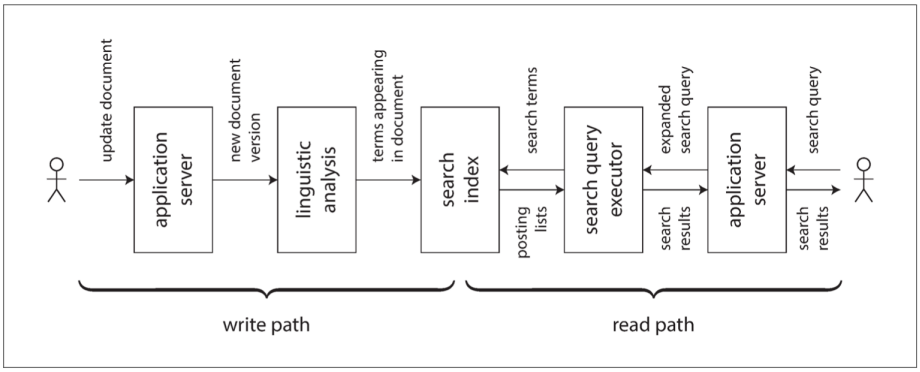

Figure 12-1. In a search index, writes (document updates) meet reads (queries).

  - Write Path: pre-computed → eagerly → eager evaluation
  - Read Path: lazy evaluation.
- **Materialized views and caching**:
  - The role of caches, indexes, and materialized views is simple: they shift the boundary between the read path and the write path.
- **Stateful, offline-capable clients**:
  - "single-page" JavaScript web apps have gained a lot of stateful capa- bilities. (Same as Mobile Apps)
    - client-side user interface interaction and
    - persistent local storage in the web browser
  - we can think of the on-device state as a cache of state on the server.
- **Pushing state changes to clients**:
  - Actively pushing states changes all the way to client devices means extending the write path all the way to the end users.
  - Each device is a small subscriber to a small stream of events.
- **End-to-end event streams**: (e.g. React, Flux, Redux)
  - Client-side state is managed by subscribing a stream of events (Event Sourcing)
  - Allow a server to push state-change events into this client-side event pipeline.
    - state changes could flow through an end-to-end write path.
  - We need to rethink the way we build our system: moving away from request/response interaction and toward publish/subscribe dataflow.
  - **keep in mind the option of subscribing to changes, not just querying the current state**.
- **Reads are events too**:
  - Writing read events to durable storage thus enables better tracking of causal dependencies
- **Multi-partition data processing**:

Aiming for Correctness

- Stateless services are much more easier to correct than Stateful;
- **The End-to-End Argument for Databases**:
  - Data system with strong safety properties can't guarantee the application is free from data loss or corruption.
    - Application bugs occur, humans make mistakes.
  - "Remove the ability of faulty code to destroy good data" → the best way to fix mistakes is to avoid it in the first place.
  - **Exactly-once execution of an operation**:
    - "Idempotent" → but some scenarios require some effort and care to make the operation "idempotent".
  - **Duplicate suppression**:
  - **Operation identifiers**:
    - In order to achieve operation idempotent, we need consider the end-to-end flow of the request. (e.g. UUID for each operation)

```
BEGIN TRANSACTION;

INSERT INTO requests
  (request_id, from_account, to_account, amount)
  VALUES('0286FDB8-D7E1-423F-B40B-792B3608036C', 4321, 1234, 11.00);

UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;

COMMIT;
```

- “Requests” table acting like an “event log”.
- **The end-to-end argument**:
  - End-to-End solution: a transaction identifier that is passed all the way from the end-user client to the database.
  - Checking data integrity: Checksums.
  - Only end-to-end encryption and authentication can protect against all of these things.
  - low-level reliability features are not by themselves sufficient to ensure end-to-end correctness
- **Applying end-to-end thinking in data systems**:
  - Currently there still no good solution to wrap low-level reliability mechanisms around high-level faults.
  - Transaction is great but expensive, especially when it comes down to heterogeneous storages.
    - It made a wide range of possible issues (concurrent writes, constraint violations, crashes, network interruptions, disk failures) and collapsed them down to two possible outcomes: **commit** or **abort**.
- **Enforcing Constraints**:
  - E.g. Uniqueness
  - **Uniqueness constraints require consensus**:
    - want to be able to immediately reject any writes that would violate the constraint, synchronous coordination is unavoidable
  - **Uniqueness in log-based messaging**:
    - “Total order broadcast” → consensus
    - Linearizable storage using total order broadcast.
  - **Multi-partition request processing**:
    - correctness can be achieved with partitioned logs, and without an atomic commit. (By starting at ‘Single-object’ writes, and then the single-object will derive other messages/events to execute.)
- **Timeliness and Integrity**:
  - **Consistency** conflates two different requirements:
    - **Timeliness**: means ensuring that users observe the system in an up-to-date state.
    - **Integrity**: means absence of corruption. (ACID → ACD)

- In general, **Integrity is much more important than timeliness**.
  - E.g. Credit Card transactions don't show within 24hrs matter far less if you got charged in the wrong amount.
- **Correctness of dataflow systems**:
  - **ACID transactions** usually provide both **timeliness** (e.g., linearizability) and **integrity** (e.g., atomic commit) guarantees.
  - Integrity is central to streaming systems.
  - Exactly-once or effectively-once semantics is a mechanism for preserving integrity.
  - reliable stream processing systems can preserve integrity without requiring distributed transactions and an atomic commit protocol. By utilize the following mechanisms:
    - Event Sourcing;
    - Deterministic derivation functions;
    - End-to-end duplicate suppression and idempotence;
    - Immutable messages.
- **Loosely interpreted constraints**:
  - many real applications can actually get away with much weaker notions of uniqueness. (e.g. user registration, back-order, over-booking etc.)
  - These applications do require integrity But they don't require timeliness on the enforcement of the constraint.
- **Coordination-avoiding data systems**:
  - **coordination-avoiding data systems** have a lot of appeal: they can achieve better performance and fault tolerance than systems that need to perform synchronous coordination.
  - It boils down to Trade-offs

- **Trust, but Verify**:
  - System models take a binary approach toward faults. Happen vs. Never Happen
    - In reality, it is about probabilities: more likely vs. less likely.
  - **Maintaining integrity in the face of software bugs**:
    - E.g. MySQL & PostgreSQL
    - Many applications don't even correctly use the features that databases offer for preserving integrity, such as foreign key or uniqueness constraints.
    - Consistency in the sense of ACID assumes everything starts off in a consistent state. (also we assume there is bug-free application code)
  - **Don't just blindly trust what they promise**:
    - Checking the integrity of data is known as **auditing**.
    - E.g. large-scale storage systems such as HDFS and Amazon S3 do not fully trust disks. (their have background processes keep checking and compare replicas)
      - Trust the disk most of the time, but not always.
    - Read → Check
  - **A culture of verification**:

    - Event-based systems can provide better auditability.
    - A deterministic and well-defined data-flow also makes it easier to debug and trace the execution of a system.
- **The end-to-end argument again**:
    - Checking the integrity of data systems is best done in an end-to-end fashion.
    - End-to-End ensured the widest coverage through the whole system(disks,networks,services and algorithms)
- **Tools for auditable data systems**: (C: Very interesting idea)
    - use **cryptographic tools** to prove the integrity of a system in a way that is robust to a wide range of hardware and software issues, and even potentially malicious actions. (e.g. Cryptocurrencies, blockchains, and distributed ledger technologies)
        - Essentially, they are distributed databases, with a data model and transaction mechanism, in which different replicas can be hosted by mutually untrusting organizations.
    - Cryptographic auditing and integrity checking often relies on **Merkle trees**.
        - **certificate transparency** is a security technology that relies on Merkle trees to check the validity of TLS/SSL certificates.

Doing the Right Thing

- Every system is built for a purpose; every action we take has both intended and unintended consequences.
- We have a responsibility to carefully consider those consequences and to consciously decide what kind of world we want to live in.
- Many datasets are about **people**: their behavior, their interests, their identity.
    - Treat users with humanity and respect.
- A technology is not good or bad in itself—what matters is how it is used and how it affects people.
- **Predictive Analytics**: (e.g. Weather, diseases, load, insurance etc.)
    - “Algorithmic prison” automated systems can systematically and arbitrarily exclude a person from participating in society without any proof of guilt, and with little chance of appeal. (C: People saying nowadays the world is more divided, maybe we can think it was caused by “algorithms”?)
    - **Bias and discrimination**:
        - “**machine learning is like money laundering for bias**”
        - Predictive analytics systems merely extrapolate from the past.
        - Data and models should be our tools, not our masters.
    - **Responsibility and accountability**:
        - Automated decision making opens the question of **responsibility and accountability**.
        - A credit score summarizes “How did you behave in the past?”

impossible.
    - Much data is statistical in nature, which means that even if the **probability distribution** on the whole is correct, individual cases may well be wrong.
        - the output of a prediction system is probabilistic and may well be wrong in individual cases.
    - A blind belief in the supremacy of data for making decisions is not only delusional, it is positively dangerous.
- **Feedback loops**:
    - Services only show you what you want to see → lead to "echo chambers" like stereotype, misinformation and polarization.
    - self-reinforcing feedback loops. → downward spiral.
    - "Systems thinking"
- **Privacy and Tracking**:
    - Data collection and tracking bring side consequences;
    - User tracking benefits users but also providing side effects.
    - "surveillance" users → the Consumer becomes the second citizen of the services/apps, but the **advertiser becomes the real user**.
    - **Surveillance**:
        - Replace "data" with "surveillance" e.g. "Designing Surveillance Intensive Applications"
    - **Consent and freedom of choice**:
        - Without understanding what happens to their data, users cannot give any meaningful consent.
        - It is not reciprocity between the services and users.
        - The relationship between the service and the user is very asymmetric and one- sided.
            - It is all set by the service provider, not the user.
    - **Privacy and use of data**:
        - Having privacy does not mean keeping everything secret; **it means having the freedom to choose which things to reveal to whom, what to make public, and what to keep secret**.
        - Privacy didn't erode but transfer to data collector;
    - **Data as assets and power**:
        - behavioral data is a byproduct of users interacting with a service → "data exhaust".
            - Or, we can think another way around, the service provider is using their service to lure users interaction/behavioral data.
        - Startups are valued by their user numbers, by "eyeballs" i.e., by their surveillance capabilities.
        - Need consider the future of the data on how it could be used or abused.
    - **Remembering the Industrial Revolution**:
        - Data is the defining feature of the information age.

privacy is the environmental challenge. — Bruce Scheneier.

- **Legislation and self-regulation**:
  - Existing law from 1995 seems totally opposite with today's need of BigData.
  - Opportunities/Risk vs. over regulation
  - Purging vs. Immutability

Summary

- No one single tool can fit all;
  - Bundle is able to help
- Data integration problem can be solved by "batch processing" and "event streams"
  - Let the data flow
- Systems of record vs. Derived data
  - Keep it loosely coupled;
- Expressing dataflows as transformations from one dataset to another also helps evolve applications.
- Unbundling the components of a database, but composing loosely coupled components when building applications
- Derived state can be updated by observing changes
  - Bring the state to the end-user device.
- Strong integrity guarantees with
  - Asynchronous event processing
  - End-to-end operation identifiers to achieve idempotent or by checking constraints async.
  - This is much better than distributed transactions
- Audits for data integrity check
- Goal, built a world that treats people with humanity and respect.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST
Designing Data Intense Application – Chapter 11: Stream Processing

NEXT POST
有关Club house 和 D & I (Diversity & Inclusion)